UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1661

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

EFFICIENT LINEAR SYSTOLIC ARRAY FOR THE KNAPSACK PROBLEM

Rumen ANDONOV
Patrice QUINTON

Avril 1992

RR-1661

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Efficient linear systolic array for the knapsack problem*
Une architecture systolique pour le problème du sac-à-dos

Rumen Andonov †
Center of Computer Science and Technology
Acad. G. Bonchev st., bl. 25-a
1113 Sofia (Bulgaria)

Patrice Quinton
IRISA, Campus de Beaulieu
35042 Rennes Cedex(France)
e-mail: quinton@irisa.fr

Projet API

Publication Interne n° 639, 18 pages - Programme 1

Mars 1992

Abstract

A processor-efficient systolic algorithm for the dynamic programming approach to the knapsack problem is presented in this paper. The algorithm is implemented on a linear systolic array where the number of cells q, the cell memory storage α and the input/output requirements are design parameters. These are independent of the problem size given by the number of the objects m and the knapsack capacity c. The time complexity of the algorithm is $\Theta(mc/q + m)$ and both the time speedup and the processor efficiency are asymptotically optimal.

*This work was partially funded by the French Coordinated Research Program C^3 and by the Esprit BRA project No 3280.

†At the present time at IRISA. Supported by a grant from the MRT.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (U.R.A. 227) UNIVERSITE DE RENNES I I.N.S.A. DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

A new procedure for the backtracking phase of the algorithm with a time complexity $\Theta(m)$ is also proposed. It is an improvement on the usual strategies used for backtracking which have a time complexity $\Theta(m+c)$.

Résumé

On présente un algorithme systolique d'efficacité élevée pour la résolution du problème du sac-à-dos par programmation dynamique. Cet algorithme est réalisé sur un réseau systolique linéaire, dont le nombre de cellules q, la mémoire des cellules α et les contraintes d'entrées/sorties sont des paramètres. Ces paramètres sont indépendants de la taille du problème, déterminée par le nombre d'objets m et la capacité c du sac-à-dos. La complexité en temps de l'algorithme est $\Theta(mc/q+m)$, et l'accélération et l'efficacité du réseau sont asymptotiquement optimales.
On propose aussi une nouvelle procédure pour la phase de retour-arrière de l'algorithme ayant une complexité $\Theta(m)$, au lieu de $\Theta(m+c)$ pour les méthodes classiques.

1 Introduction

Suppose that m types of objects are being considered for inclusion in a knapsack of capacity c. For $i = 1, 2, \ldots, m$, let p_i be the unit value and w_i the unit weight of the i-th type of object. The values w_i, p_i, $i = 1, 2, \ldots, m$, and c are all positive integers. The problem is to find out the maximum total profit without exceeding the capacity constraint, i.e.

$$\max \{\sum_{i=1}^{m} p_i z_i : \sum_{i=1}^{m} w_i z_i \leq c, z_i \geq 0 \text{ integer}, \ i = 1, 2, \ldots, m\}, \tag{1}$$

where z_i is the number of i-th type objects included in the knapsack. This is a classical combinatorial optimization problem with a wide range of application (see Garfinkel and Nemhauser [1], Hu [2], Martello and Toth [3]). Sometimes (1) is referred to as the integral knapsack problem (see Teng [4]), sometimes as the unbounded knapsack problem (see, e.g.Martello and Toth [3]). We shall call it simply the knapsack problem. If additional constrains $z_i \in \{0, 1\}, i = 1, 2, \ldots, m$ are added to (1), then the restricted problem is called the 0/1 knapsack problem. Problem (1) is well known to be NP-*hard* (see Martello and Toth [3]). However it is known that this problem can be solved sequentially in $O(mc)$ time. This time bound is not polynomial in the size of the input since $\log_2 c$ bits are required to encode the input c. Such a time bound is called *pseudo-polynomial* time [5].

With the advent of parallel processors many researchers concentrated their efforts on the development of efficient parallel algorithms for solving the knapsack and the 0/1 knapsack problems. Dynamic programming [6, 7, 8, 4], branch-and-bound [9, 10] and approximate algorithms [11, 12] are the most popular combinatorial optimization techniques for these problems. The number of processors, required by most of the parallel algorithms, is exponential in the size of the input and these algorithms have a very low processor efficiency. For example the best time complexity algorithm for (1) proposed by Teng in [4] requires $M(c)$ processors to solve the problem in $O(log^2(mc))$ time. The function $M(n)$ above denotes the

number of processors needed for multiplying two n by n matrices in $O(logn)$ parallel time. It is known that $n^2 \leq M(n) \leq n^3$. Therefore the knapsack capacity c is a factor in the processor complexity of the algorithm and $1/c$ is a factor in its efficiency which approaches zero as c increases.

In this paper we concentrate on one of the most conventional approaches for solving the knapsack problems - dynamic programming. This approach is based on the *principle of optimality* of Bellman [13] and usually contains two phases. In the first (forward) phase the maximum value of the objective function is computed, i.e. the value $f_m(c)$ such that $f_m(c) = \max \{\sum_{i=1}^{m} p_i z_i : \sum_{i=1}^{m} w_i z_i \leq c,\ z_i \geq 0 \ \text{integer}, i = 1, 2, \ldots, m\}$. In the second (backtracking) phase the integers $z_i^*, i = 1, 2, \ldots, m$, such that $\sum_{i=1}^{m} p_i z_i^* = f_m(c)$ are found.

Recently, a dynamic programming algorithm for the 0/1 knapsack problem, which may run on any number of processors available, was presented in the work of Lin and Storer [8]. Its running time is $O(mc/q)$ on an EREW PRAM of q processors and this algorithm has optimal time speedup and processor efficiency.

In [7] a pipeline architecture containing a linear array of q processors, queue and memory modules is proposed for the knapsack problem by Chen, Chern and Jang. This architecture allows one to achieve an optimal speedup of the algorithm which has a time complexity $O(mc/q + m)$ and an efficiency $E = \Theta(1/(1 + 1/qc))$ which approaches $\Theta(1)$ as c increases.

Our work belongs to another branch of research which is related to the design of systolic arrays for dynamic programming problems. The difficulties in this field of research arise from the necessity for the algorithms to meet the requirements of modularity, ease of layout, simplicity of communication and control and scalability. Dynamic programming is a powerful optimization methodology which is worthy of study for its suitability for VLSI systolic implementation. It was considered in numerous works (see Guibas, Kung and Thompson [14], Bitz and Kung [15], Myoupo [16]). For the 0/1 knapsack problem this approach was considered in the work of Li and Wa [17] and of Lipton and Lopresti [18]. The results obtained are not applicable in the case of the knapsack problem due to a peculiarity in its recurrent formulation. An attempt to design fixed-size modular linear systolic array (a ring with buffer memory between the last and the first cells) for (1) appeared recently in the paper of Andonov, Aleksandrov and Benaini [19]. The running time of the presented algorithm is $\Theta(mc^2/q\alpha + c + m)$ on q cells, each of storage capacity α, where α is a design parameter. To the knowledge of the authors, this is an unique problem size independent algorithm for the knapsack problem. It has the drawback that its processor efficiency approaches zero as c increases.

A new dynamic programming implementation for the knapsack problem which runs on any number of processors is presented in this paper. The architecture is similar to that of [19] - a ring containing a buffer memory and q identical cells, each with a local addressable memory of size α. A new algorithm for the backtracking phase with time complexity $\Theta(m)$ is proposed. It improves on the usual strategies used for backtracking in the knapsack problem (see Hu [2] and Garfinkel and Nemhauser [1]) which have a time complexity $\Theta(m + c)$. Thus this phase of the dynamic programming approach, which is sequential, becomes independent of the parameter c. This algorithm allows also a pure linear systolic array to

be designed for both phases of the dynamic programming method (i.e. the forward and the backtracking phase). The problem size memory independence of the algorithm is provided, i.e. the algorithm is designed under the requirement that the cell memory capacity does not depend on the problem size. Hence our algorithm meets all the requirements for VLSI systolic implementation as the algorithm proposed by Andonov, Aleksandrov and Benaini [19] but substantially improves on its speedup and efficiency. More precisely, for the running time T_q on q processor, each one of memory size α we obtain to within a constant factor $mw_{min}(c+q)/q\alpha + m \leq T_q \leq mw_{max}(c+q)/q\alpha + m$, where $w_{max} = \max_{i=1}^{m}\{w_i\}, w_{min} = \min_{i=1}^{m}\{w_i\}, \alpha \leq w_{max}$ and $q \leq mw_{max}/\alpha$. This result confirms once again the observation (see Teng [4] and Andonov and Gruau [20, 21]) that w_{max} and w_{min} are important parameters for the knapsack problem. These influence the running time of the parallel algorithm in contrast to the serial algorithm which depends on m and c only. It is well known that in many practical applications w_{max} is much less than the knapsack capacity c, i.e., $w_{max} \ll c$. A reasonable assumption is that w_{max} and w_{min} are constant factors as c increases. This observation implies that the speedup and the efficiency both approach their optimal values respectively $\Theta(q)$ and $\Theta(1)$ as c increases. Therefore we obtain the same speedup and efficiency as in the paper of Chen, Chern and Jang [7]. Note that the algorithm in [7] is not VLSI oriented since it runs exclusively under the assumption that the memory capacity of any processor is at least equal to the knapsack capacity c, i.e. $\alpha \geq c$.

This paper is organized in the following way. Section 2 describes both phases of the dynamic programming approach for problem (1) on a serial machine. Section 3 presents our new algorithm for backtracking and analyses its complexity. In section 4 we discuss our parallel implementation on a linear array containing unbounded number of cells of storage capacity α. In section 5 the same approach is developed on a ring with fixed number of cells.

2 Dynamic programming approach for the knapsack problem

In this section we present the dynamic programming approach for the knapsack problem on a serial machine. It consists of two phases - the forward and the backtracking phase.

2.1 Forward phase

Let $f_k(j)$ be the maximum value that can be achieved in (1) from a knapsack of size j, $0 \leq j \leq c$, using only the first k types of objects, $1 \leq k \leq m$. That is

$$f_k(j) = \max\{\sum_{i=1}^{k} p_i z_i : \sum_{i=1}^{k} w_i z_i \leq j, z_i \geq 0 \text{ integer}, i = 1, 2, \ldots, k\}. \quad (2)$$

The principle of optimality [13, 1] states that for $\forall k, 1 \leq k \leq m$ and $\forall j, 0 \leq j \leq c$ we have :

$$f_k(j) = \max\{f_{k-1}(j), f_k(j - w_k) + p_k\}. \quad (3)$$

The optimal value of (1) $f_m(c)$, can be found in m stages by generating successively the functions $f_1, f_2, \ldots, f_m$ using equation (3) and the initial conditions $f_0(j) = 0, f_k(0) = 0$ and $f_k(i) = -\infty$ for $1 \leq k \leq m, 0 \leq j \leq c$ and $i < 0$. By stage k we shall denote the computation of all the values of the function f_k. This approach solves simultaneously a set of knapsack problems with knapsack capacities $1, 2, \ldots, c$.

Any serial algorithm based on recurrent equation (3) requires $O(mc)$ time to find the optimal value $f_m(c)$. Furthermore, this is optimal, since $\Omega(mc)$ time is needed to compute $f_m(c)$ in the case $w_i = 1, i = 1, 2, \ldots, m$ because any value $f_k(j), 1 \leq k \leq m, 1 \leq j \leq c$, depends on its previous value $f_k(j-1)$. Therefore the time complexity of the serial algorithm which finds $f_m(c)$ through the equation (3) is $\Theta(mc)$. This is the best existing serial algorithm and we shall use its running time for determine the *speedup* and the *efficiency* of our parallel algorithm.

The communication required for the execution of equation (3) can be described by means of directed graph, called the *dependence graph (DG)*. Let $\mathbf{N}$ denote the set of natural numbers, i.e. $\mathbf{N}=\{0, 1, 2, \ldots\}$. Let $\mathcal{G} = (\mathcal{D}, \mathcal{A})$ be the *DG* for equation (3), where $\mathcal{D} = \{(j,k) \in \mathbf{N}^2 : 0 \leq j \leq c, 1 \leq k \leq m,\}$ is the set of nodes and $\mathcal{A}$ is the set of directed arcs. Each node $(j,k) \in \mathcal{D}$ of the *DG* represents an operation performed by the algorithm and the arcs are used to represent data dependencies. For example figure 1 (a) depicts the *DG* for a knapsack problem, with $m = 3, c = 6, w_1 = 4, w_2 = 3, w_3 = 2$. Each node (j,k) represents one calculation, detailed in figure 1 (b). The peculiarity of this graph is that in any column the dependence vectors depend on the weights $w_i, i = 1, 2, \ldots, m$. Such a dependency is not a *uniform* dependency. This peculiarity makes the knapsack recurrence equation difficult to transform into a systolic array using the well-known dependence mapping approach (see Quinton and Robert, [22]).

As noticed in [7], the following properties of equation (3) make it suitable for pipelined computation:

1. In each stage the same operations are performed.
2. Data dependencies occur in two cases only: between adjacent stages and within the same stage.
3. If $f_k(j), j = 0, 1, \ldots, c$, are computed in increasing order of j, then $f_k(j)$ can be computed whenever the values $f_{k-1}(j), j = 0, 1, \ldots, c$, are available.

2.2 The classical backtracking algorithm

An approach to find the solution vector $z^* \in \mathbf{N}^m$ such that $\sum_{i=1}^{m} p_i z_i^* = f_m(c)$ is discussed in this section. It is based on the work of Hu [2].

In the course of the forward phase a pointer $u_k(j)$ is associated to any value $f_k(j), (j,k) \in \mathcal{D}$ in such a way that $u_k(j)$ is the index of the last type of object used in $f_k(j)$. In other words if $u_k(j) = r$ this means $z_r \geq 1$, or the r-th object is used in $f_k(j)$ and $z_l = 0$ for all $l > r$. The value $u_k(j)$ is used to keep the history of the first dynamic programming phase.

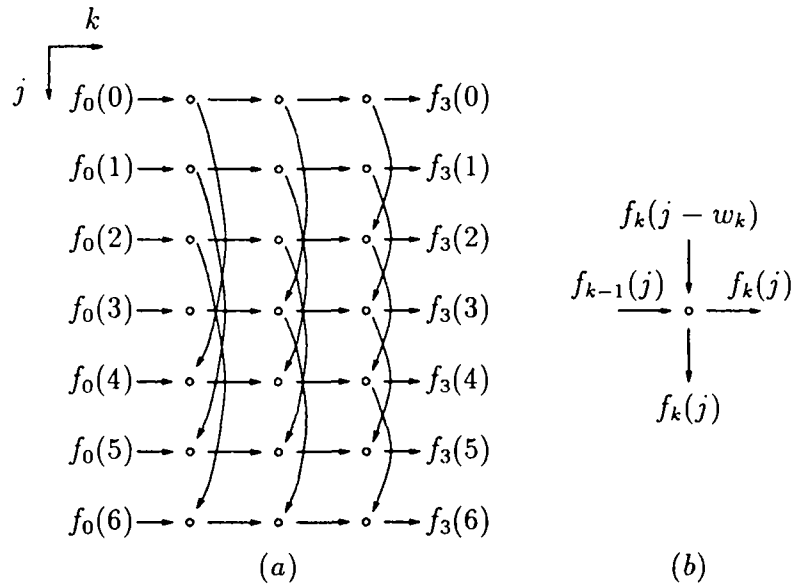


Figure 1: The dependence graph for the given example

The boundary conditions for $u_k(j)$ are

$$\forall j : 0 \leq j \leq c : u_1(j) = \begin{cases} 0 & \text{if } f_1(j) = 0 \\ 1 & \text{if } f_1(j) \neq 0. \end{cases}$$

In general we set

$$\forall k, \forall j : 1 < k \leq m, \quad 0 \leq j \leq c \quad : u_k(j) = \begin{cases} k & \text{if } f_k(j - w_k) + p_k > f_{k-1}(j) \\ u_{k-1}(j) & \text{otherwise.} \end{cases} \tag{4}$$

As shown in [2], definition (4) allows the solution vector $z^* \in \mathbf{N}^m$ of the knapsack problem to be found from the values of the function u_m only. The following algorithm can be used for this purpose:

Hu's backtracking algorithm

```
j := c;
for k = m downto 1 do
z*_k = 0
    while u_m(j) = k do
                begin z*_k := z*_k + 1;
                    j := j - w_k
```

end

This algorithm has time and space complexity $T = \Theta(m + c)$ and $S = \Theta(c + m)$ respectively and it is the best known strategy. Another strategy with time complexity $T = \Theta(m+c)$ and space complexity $S = \Theta(mc)$ is presented by Garfinkel and Nemhauser in [1].

The backtracking phase is sequential. We cannot find $z_k^*, 1 \leq k < m$, until z_{k+1}^* is computed, (i.e this phase requires at least $\Omega(m)$ operations). In the next section we show how the algorithm above can be modified in order to reach this bound.

3 A modified backtracking algorithm

If we associate a value p_k to any vertical arc $((i,k),(i + w_k, k))$ in column k of the DG for problem (1) then the value $f_n(j)$ for any pair $(j,n) \in \mathcal{D}$ can be regarded as the value of the optimal path $(S_1, S_2, \ldots, S_n)$ from the first stage to the n-th one. We denote by $S_k, k = 1,2,\ldots,n$, the subpath of the optimal path in column k. The elements of S_k are of the form $((i,k),(i + w_k,k),\ldots,(i + w_k z_k, k))$, $0 \leq i \leq j - w_k z_k$, where z_k are such that $\sum_{k=1}^{n} p_k z_k = f_n(j)$. Let I_k denote the sequence of the first indices of the elements of the optimal subpath S_k in column k, i.e.

$$I_k = \{i : (i,k) \in S_k\}.$$

Let i_{min}^k denote the minimum index in I_k, i.e.

$$i_{min}^k = \min\{i : i \in I_k\}.$$

By the definition of the function u_k we have

$$u_k(i_{min}^k) < k$$

and

$$\forall i \in I_k : i \neq i_{min}^k \Rightarrow u_k(i) = k.$$

The idea of the modified backtracking algorithm is for any $(j,k) \in \mathcal{D}$ to keep a record of the corresponding value i_{min}^k. It is more suitable in our algorithm to save the values $i_{min}^k + w_k$. In order to generate these values during the forward phase we introduce the function v_k defined as follows:

$$v_k(j) = \begin{cases} v_{k-1}(j) & \text{if } u_k(j) < k \\ j & \text{if } u_k(j) = k \quad \text{and } u_k(j - w_k) < k \\ v_k(j - w_k) & \text{if } u_k(j) = k \quad \text{and } u_k(j - w_k) = k \end{cases} \tag{5}$$

for $\forall k, \forall j : 1 \leq k \leq m, 0 \leq j \leq c$ and $v_k(j) = 0$ for $k = 0, 0 \leq j \leq c$.

When the values of $v_k, k = 1, 2, \ldots, m$, are found, we can easily trace the values z_k that yield $f_k(j)$ for any $j \in I_k$ by the formula

$$z_k = \begin{cases} 0 & \text{if } u_k(j) < k \\ (j - v_k(j))/w_k + 1 & \text{if } u_k(j) = k. \end{cases} \quad (6)$$

In fact (6) determines how many units of object k are used in $f_k(j)$.

Once the value z_k has been computed, the total weight limitation j is reduced to $j_{new} = v_k(j) - w_k$, for which the inequality $u_k(j_{new}) < k$ holds and therefore $u_k(j_{new}) = u_{k-1}(j_{new})$. To find the value z_{k-1} we proceed to check the values $u_{k-1}(j_{new})$ and $v_{k-1}(j_{new})$ in the same way.

The functions $v_k, k = 1, 2, \ldots, m$ are similar to the functions u_k by their properties. For $\forall j : u_k(j) < k$ the function v_k keeps the values of the functions v_{k-1}. This property allows the values $z_k, k = 1, 2, \ldots, m$ to be found from the values of the function u_m and v_m only. The following algorithm is used.

Modified backtracking algorithm

```
j := c;
for k = m downto 1 do
    if u_m(j) < k then z*_k = 0
                  else
                  begin z*_k := (j - v_m(j))/w_k + 1;
                        j := v_m(j) - w_k
                  end
```

In this way the values of u_m are used to move back along the k axis of the DG in figure 1. The values of v_m are used to move back along the j axis. Since the computation of z_k^* for any $k, 1 \leq k \leq m$ requires $\Theta(1)$ operations then we obtain the following property.

Corollary 1 *The modified backtracking algorithm has time and space complexity* $T = \Theta(m)$ *and* $S = \Theta(c + m)$ *respectively.*

The computation of the functions $v_k, k = 1, 2, \ldots, m$ can be done simultaneously with the computation of the functions f_k. Comparing (5) and (4) we see that the time complexity to compute the values of v_k is the same as the time complexity to compute the values of u_k. More details concerning the implementation of the modification proposed are discussed in the next section.

4 Linear systolic arrays

In this section we consider the implementation of the knapsack algorithm in linear arrays composed of q identical cells $C_k, k = 1, 2, \ldots, q$, where $q \geq m$. Each cell C_k has two addressable memories F_k and V_k each of size α, where α is a design parameter. The purpose of F_k

and V_k is to save the values of the functions f_k and v_k respectively. We consider successively two cases: $\alpha \geq w_k$ for $\forall k, 1 \leq k \leq m$, and $\exists k, 1 \leq k \leq m : w_k > \alpha$. In the first case a straightforward projection of the dependence graph of figure 1 along the j axis provides a systolic array of m cells. This case is described in section 4.1, where we also show how the backward phase can be implemented on the same array. This array is said to be *non modular*, as it cannot accommodate a problem of any size. In the second case it is possible to implement the algorithm on a linear systolic array, by replacing cell C_k of the non modular design by so-called *macro-cells* comprising $\lceil w_k/\alpha \rceil$ cells. Section 4.2 is devoted to this case.

4.1 Non modular systolic array

Forward phase The operation of the systolic array is best explained using the dependence projection method (see [23], chapter 12), which amounts to scheduling the dependence graph and projecting it along a conveniently chosen direction. A *timing function* is a mapping $t : \mathcal{D} \rightarrow N$, such that if the computation on vertex $v \in \mathcal{D}$ depends on vertex $w \in \mathcal{D}$, then $t(v) > t(w)$. An *allocation function* is a mapping $a : \mathcal{D} \rightarrow [1, q]$, such that a($v$) is the number of the processor that executes the calculations attached to vertex $v \in \mathcal{D}$. The mapping a must be chosen in such a way that a processor has no more than one calculation to perform at a given instant. In addition to the previous well known functions, we use the mapping $addr : \mathcal{D} \rightarrow [1, \alpha]$, defined in such a way that addr(v) is the number of the memory location in processor a(v) where data $v \in \mathcal{D}$ is stored.

Obviously, the function $t(j, k) = j + k$ is a timing-function for the dependence graph of figure 1. An allocation function $a(j, k) = k$ corresponds to a projection of the dependence graph along axis j. It yields a linear unidirectional systolic array of m cells. Since for any k, the values $f_k(j), j = 0, 1, \ldots, c$ are computed sequentially and $f_k(j)$ depends on $f_k(j - w_k)$, then the value $f_k(j)$ can be stored in the same memory location in cell C_k as the value $f_k(j - w_k)$. We assume that the memory size α meets $\alpha \geq w_k$, for any $1 \leq k \leq m$. Therefore the address of any data $(j, k) \in \mathcal{D}$ in F_k is addr$(j, k) = 1 + j \bmod w_k$. The total time for the forward phase is

$$t(c, m) = c + m. \tag{7}$$

Backtracking phase The values $u_k(j)$ and $v_k(j)$ are computed and propagated through the considered array simultaneously with the value $f_k(j)$. The computation of the function u_k does not require a supplementary local memory. We show in this section that a memory V_k of size w_k in cell C_k is enough for the computations of the function v_k.

Notice that if two nodes (s, k), (t, k) of the DG belong to the same subpath in a column k then $s \bmod w_k = t \bmod w_k$. Two subpaths S_1 and S_2 of a column k are said to be *dependent* if $s \bmod w_k = t \bmod w_k$ for $(s, k) \in S_1$ and $(t, k) \in S_2$. As with the values of the function f_k, the values $v_k(j), j = 0, 1, \ldots, c$ are computed sequentially and $v_k(j)$ depends on $v_k(j - w_k)$, i.e. the value $v_k(j)$ can be stored in the same memory location in V_k as the value $v_k(j - w_k)$. The difficulty arises from the observation that the dependent paths in column k are projected in the same location in V_k. To overcome this problem we denote the beginning of a new path

in column k by the following formula:

$$V_k(j) = \begin{cases} -1 & \text{if } f_{k-1}(j) \geq f_k(j-w_k) + p_k \\ j & \text{if } f_{k-1}(j) < f_k(j-w_k) + p_k \quad \text{and } V_k(j-w_k) = -1 \\ V_k(j-w_k) & \text{if } f_{k-1}(j) < f_k(j-w_k) + p_k \quad \text{and } V_k(j-w_k) \neq -1 \end{cases} \tag{8}$$

Then the value $v_k(j)$ is defined by

$$v_k(j) = \begin{cases} v_{k-1}(j) & \text{if } f_{k-1}(j) \geq f_k(j-w_k) + p_k \\ V_k(j) & \text{otherwise .} \end{cases} \tag{9}$$

Since $v_k(j)$ are computed sequentially for $j = 0, 1, \ldots, c$ then the values $V_k(j)$ and $V_k(j - w_k)$ can be stored in the same location $\text{addr}(j,k) = 1 + j \bmod w_k$.

It is easily seen that (9) is equivalent to (5). Definition (9) has the advantage of being independent of the function u_k and overcomes the problem when two dependent paths in column k are projected in the same memory location.

The values of the functions u_m and v_m leave the last cell C_m and are stored in a supplementary memory of size $2c$. Then the modified backtracking algorithm can be executed by the host computer (or by the last cell C_m if it is connected bidirectionally with the supplementary memory).

4.2 Modular systolic array

Let us assume now that the memory size α of the cell is independent of the

problem and there exists k, such that $w_k > \alpha$. The idea is to emulate the operation of each elementary cell C_k of the previous linear design by one macro-cell MC_k which is composed of $\lceil w_k/\alpha \rceil$ cells (see figure 2), each one with a memory of size α.

The mapping of the dependence graph is made as follows. Any computation $f_k(j)$ is performed on macrocell MC_k. The value $\text{addr}(j,k) = 1 + j \bmod w_k$ defined in section 4.1 is the address where the value $f_k(j)$ would be stored in MC_k, if α is large enough. Here, the memory w_k is emulated by the memory of the $\lceil w_k/\alpha \rceil$ subcells. To know where $f_k(j)$ is stored in MC_k, we associate with $\text{addr}(j,k)$ a pair $(\text{sc}(j,k), \text{addr}'(j,k))$ which gives respectively the subcell where $f_k(j)$ is stored, and the address in the memory of this subcell. We thus have:

$$\begin{aligned} \text{sc}(j,k) &= \lceil \text{addr}(j,k)/\alpha \rceil \\ \text{addr}'(j,k) &= \text{addr}(j,k) \bmod \alpha. \end{aligned}$$

In this way $f_k(j)$ and $f_k(j - w_k)$ are stored in the same address of the same cell. The calculations executed by a subcell of MC_k is deduced by the following rules. Macro-cell MC_k processes in sequence the calculations associated with nodes (j,k), $0 \leq j \leq c$, of the dependence graph. When a subcell of MC_k receives data which corresponds to $f_{k-1}(j)$, it processes the calculation associated to this value if $f_k(j)$ is stored in its memory, otherwise, it

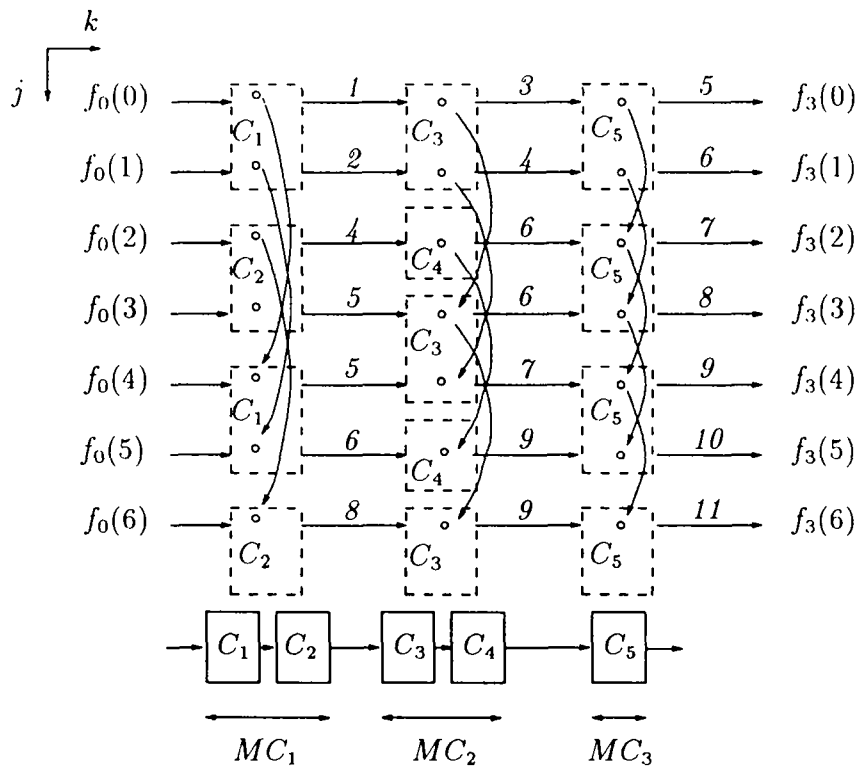


Figure 2: Example of modular mapping for $\alpha = 2$

sends this value unchanged to the next subcell. In this way a data $f_{k-1}(j)$ crosses $sc(j,k)-1$ cells in MC_k without being processed. On the other hand, once computed any value $f_k(j)$ is transmitted forward in the array $\lceil w_k/\alpha \rceil - \mathrm{sc}(j,k)$ subcells in order to leave macrocell MC_k. Thus, the new timing-function is given by the following recurrence :

$$t(j,k) = \begin{cases} j + \mathrm{sc}(j,k) & \text{if } k = 1 \\ t(j,k-1) + \lceil w_{k-1}/\alpha \rceil - \mathrm{sc}(j,k-1) + \mathrm{sc}(j,k) & \text{otherwise.} \end{cases} \quad (10)$$

The values of the timing-function for the given example are depicted on the right hand of the corresponding vertices in figure 2.

The operation of one subcell is now described. The value $f_k(j)$ is computed by the subcell $\mathrm{sc}(j,k)$ of the macrocell MC_k and is stored in memory location $\mathrm{addr}'(j,k)$. All the other cells of the macrocell MC_k merely transmit the value $f_k(j)$. To realize this, we associate a counter $cell(j,k)$ with any data $f_k(j)$. This counter gives the number of elementary cells the associated data has to be propagated in order to reach the cell where the value $f_{k+1}(j)$ is computed. According to (10) the counter can be obtained by the formula

$$cell(j,k) = \begin{cases} \mathrm{sc}(j,k+1) & \text{if } k = 0 \\ \lceil w_k/\alpha \rceil - \mathrm{sc}(j,k) + \mathrm{sc}(j,k+1) & \text{otherwise.} \end{cases} \quad (11)$$

From (11) the new allocation function follows easily:

$$a(j,k) = \begin{cases} \mathrm{sc}(j,k) & \text{if } k = 1 \\ a(j,k-1) + cell(j,k-1) & \text{otherwise.} \end{cases} \quad (12)$$

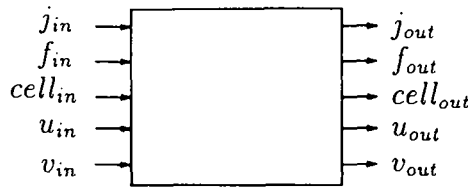
During the computations of $f_k(j)$ each cell generates, in addition, the values $v_k(j)$ and $u_k(j)$ needed for the backtracking, as was shown in the previous section. The structure and the program for the forward phase of the cell are depicted in figure 3.

To realize this algorithm each subcell of the macrocell MC_k keeps the values p_k, w_k, w_{k+1}, k in registers $p, w, wnext, obj$ respectively. The data input into the leftmost cell during the first $c+1$ instants t, $t = 0, 1, \ldots, c$ are $f_{in} = 0, u_{in} = 0, v_{in} = 0, j_{in} = t$, $\mathrm{cell}_{in} = \lceil (1+t \bmod w_1)/\alpha \rceil$. According to (10) the rightmost cell of the array outputs the value $f_m(c)$ in time $t(c,m) = c + \sum_{i=1}^{m} \lceil w_i/\alpha \rceil$.

From the above discussion we are led to the following result:

Proposition 1 *The algorithm for the modular systolic array requires* $c + \sum_{i=1}^{m} \lceil w_i/\alpha \rceil$ *time to find* $f_m(c)$ *on* $\sum_{i=1}^{m} \lceil w_i/\alpha \rceil$ *processors, each with storage capacity* α.

Corollary 2 *If* $\alpha \geq w_i, i = 1, 2, \ldots, m$ *then the algorithm presented here requires* $c+m$ *time to find* $f_m(c)$ *on* m *processors.*




```
repeat
case
   cell_in = 1  → [  {computation}
                     addr  := 1 + j_in mod w;
                     addr' := addr mod α;
                     {compute f_k(j) and store in F}
                     if j_in < w then  F(addr') := f_in
                                 else  F(addr') := max{f_in, F(addr') + p}
                     fi;
                     {compute u_k(j) }
                     if F(addr') ≤ f_in then  u_out := u_in
                                        else  u_out := obj
                     fi;
                     {compute v_k(j) and store in V}
                     if F(addr') ≤ f_in then V(addr') := -1
                                        else
                                             if V(addr') = -1 then V(addr') := j_in
                                             fi
                     fi;
                     if F(addr') ≤ f_in then  v_out := v_in
                                        else  v_out := V(addr')
                     fi;
                     {number of cells to be skipped by f_k(j)}
                     cell_out := ⌈w/α⌉ - ⌈addr/α⌉ + ⌈(1 + j_in mod wnext)/α⌉;
                     j_out  := j_in;
                     f_out  := F(addr')
                     ]
   cell_in > 1  → [  {transmit}
                     cell_out  := cell_in - 1;
                     j_out  := j_in;
                     f_out  := f_in
                     ]
end_case
end_repeat
```

Figure 3: The basic cell and its operation during the forward phase

5 A Ring

Obviously, the first cell of the linear array from the previous section becomes idle in c time. On the other hand, any data needs $t_{cross} = \sum_{i=1}^{m} \lceil w_i/\alpha \rceil$ time to cross the array and to leave the last cell. Therefore, if $t_{cross} > c$, then the first cell can be reused, instead of adding new cells to the array. This idea can be easily extended in the case where we aim to bound the number of the cells used to some fixed number q, where $q \leq t_{cross} < c$. The solution is to add a buffer memory module Q to the array. This memory receives the data from the cell C_q, stores it and sends it to the cell C_1 when this cell becomes idle. The memory Q also saves the vectors u_m and v_m at the end of the forward phase. Therefore a necessary condition for the ring to be able to solve problem (1) is that the size of Q is at least $2m+4c$. The resulting architecture (see figure 4) is called Ring and is well known from systolic computation [24].

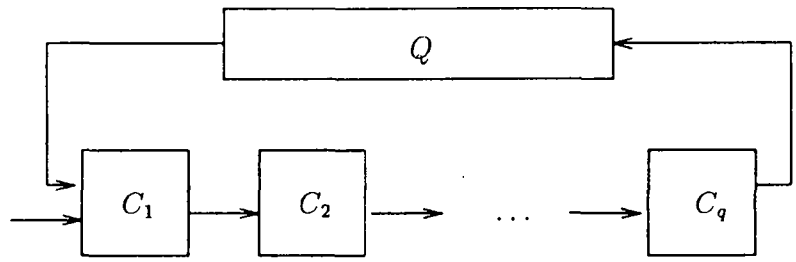


Figure 4: The Ring

As was shown, the algorithm in the previous section needs t_{cross} cells, each with storage capacity α to solve the knapsack problem. Therefore, the linear array in the ring in figure 4 has to be used $\lceil t_{cross}/q \rceil$ times to solve it. For this purpose, the set $f_k(j), k = 1, 2, \ldots, m, j = 0, 1, \ldots, c$ is partitioned into $\lceil t_{cross}/q \rceil$ bands and each one of these bands can be executed individually by the linear processor array. Let B_i denote the ith band. The partitioning is performed as follows:

$$f_k(j) \in B_i \Leftrightarrow \lceil a(j,k)/q \rceil = i,$$

where $a(j,k)$ is defined in (12).

Since c values are input in any cell and a data needs q time to pass through C_1 to C_q, then a new set of data (a new band) can be input to the cell C_1 any $\max\{c,q\} = \Theta(c+q)$ time. Obviously, the inequality

$$m\lceil w_{min}/\alpha \rceil \leq t_{cross} \leq m\lceil w_{max}/\alpha \rceil, \tag{13}$$

where $w_{max} = \max_{i=1}^{m}\{w_i\}, w_{min} = \min_{i=1}^{m}\{w_i\}$, is satisfied. Hence, for the running time T_{for} of the parallel forward dynamic programming part we obtain to within a constant factor

$$mw_{min}(c+q)/q\alpha \leq T_{for} \leq mw_{max}(c+q)/q\alpha, . \tag{14}$$

For the total running time T_q we have to add to T_{for} the time needed to load the coefficients $p_i, w_i, i = 1, 2, \ldots, m$ into the cells and the time for the backtracking dynamic programming part, which can be done in additional $\Theta(m)$ time. Therefore, for the total running time T_q from (14) we obtain to within a constant factor

$$mcw_{min}/q\alpha + mw_{min}/\alpha + m \leq T_q \leq mcw_{max}/q\alpha + mw_{max}/\alpha + m.$$

A reasonable assumption is that w_{max} and w_{min} do not change as c increases and therefore α/w_{max} and α/w_{min} are constant factors. Hence, we obtain for the total time complexity of our algorithm on the ring

$$T_q = \Theta(mc/q + m). \tag{15}$$

The speedup and the efficiency of the algorithm both approach their optimal values respectively $\Theta(q)$ and $\Theta(1)$ as c increases.

6 Conclusion

Dynamic programming is an effective recursive approach widely used in optimization problems. The results of numerous researchers show that this approach is very suitable for systolic implementation. One interesting exception was the knapsack problem whose dependence graph depends on the weights of the considered objects and lacks the regularity usually required for obtaining a systolic implementation. A method for synthesis of systolic arrays for such class of problems is not yet developed.

In this paper, a previously reported linear systolic array for the knapsack problem is transformed into a more efficient structure. To get this solution, an optimal data partitioning for both the forward and the backtracking dynamic programming phases is proposed. The design parameters (the number of the cells and their memory storage) are problem size independent. Due to its simplicity, regularity and local interconnections the array is suitable for VLSI implementation. To the knowledge of the authors the proposed algorithm is the first processor-efficient VLSI oriented algorithm for the knapsack problem.

Acknowledgements

The authors would like to thank Liam Marnane, Fédéric Raimbault and Nicola Yanev for many helpful discussions. Frédéric Raimbault also wrote a simulation program of the algorithm using the systolic language RELACS.

References

[1] R. Garfinkel and G. Nemhauser, *Integer Programming.* John Wiley and Sons, 1972.

[2] T. C. Hu, *Combinatorial Algorithms.* Addison-Wesley Publishing Company, 1982.

[3] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementation.* John Wiley and Sons, 1990.

[4] S. Teng, "Adaptive parallel algorithm for integral knapsack problems," *J. of Parallel and Distributed Computing*, vol. 8, pp. 400–406, 1990.

[5] M. Garey and D. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[6] J. Lee, E. Shragowitz, and S. Sahni, "A hypercube algorithm for the 0/1 knapsack problems," *J. of Parallel and Distributed Computing*, vol. 5, pp. 438–456, 1988.

[7] G. Chen, M. Chern, and J. Jang, "Pipeline architectures for dynamic programming algorithms," *Parallel Computing*, vol. 13, pp. 111–117, 1990.

[8] J. Lin and J. A. Storer, "Processor-efficient hypercube algorithm for the knapsack problem," *J. of Parallel and Distributed Computing*, vol. 13, pp. 332–337, 1991.

[9] T. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *CACM*, vol. 27, no. 6, 1984.

[10] J. Jansen and F.M.Sijstermans, "Parallel branch-and-bound algorithms," *Future Generation Computer Systems*, vol. 4, pp. 271–279, 1988.

[11] P. Gopalakrishnan, I. Ramakrishnan, and L. Kanal, "Approximate algorithms for the knapsack problem on parallel compurers," *Information and Computation*, vol. 91, pp. 155–171, 1991.

[12] T. E. Gerasch, "A parallel approximation algorithm for 0/1 knapsack," in *Proc. International Conference on Parallel Processing*, pp. 302–303, 1991.

[13] R. Bellman, *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[14] L.Guibas, H. Kung, and C. Thompson, "Direct VLSI implementation of combinatorial algorithms," in *Proc. 1979 Caltech. Conf. on VLSI*, pp. 509–525, 1979.

[15] F. Bitz and H. T. Kung, "Path planning on the warp computer: using a linear systolic array in dynamic programming," *Int.J. Computer Math.*, vol. 25, pp. 173–188, 1988.

[16] J. Myoupo, "A fully pipelined solutions constructor for dynamic programming problems," in *Advances in Computing - ICCI'91, Proc. Inter. Conf. Comput. Inform.*, Springer-Verlag, 1991. Lecture Notes in Computer Science 497.

[17] G. Li and B. Wa, "Systolic processing for dynamic programming problems," in *Proc. International Conference on Parallel Processing*, pp. 434–441, 1985.

[18] R. J. Lipton and D. Lopresti, "Delta transformation to symplify VLSI processor arrays for serial dynamic programming," in *Proc. International Conference on Parallel Processing*, pp. 917–920, 1986.

[19] R. Andonov, V. Aleksandrov, and A. Benaini, "A linear systolic array for the knapsack problem," Tech. Rep., Center of Computer Science and Technology, Acad. G. Bonchev st., bl. 25a, Sofia 1113, Bulgaria, 1991.

[20] R. Andonov and F. Gruau, "A 2D toroidal systolic array for the knapsack problem," in *Algorithms and Parallel VLSI Architectures II,* (Bonas, France), June 1991.

[21] R. Andonov and F. Gruau, "A 2D modular toroidal systolic array for the knapsack problem," in *ASAP'91,*(Barcelona, Spain), September 1991.

[22] P. Quinton and Y. Robert, *Algorithmes et architectures systoliques.* Masson, 1989. English translation by Prentice Hall, *Systolic Algorithms and Architectures*, Sept. 1991.

[23] C. Dezan, E. Gautrin, H. Le Verge, P. Quinton, and Y. Saouter, "Synthesis of systolic arrays by equation transformations," in *ASAP'91*, (Barcelona, Spain), IEEE, September 1991.

[24] H. T. Kung, "Why systolic architectures," *Computer*, vol. 15, pp. 37–46, 1982.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 630 EREBUS, A DEBUGGER FOR ASYNCHRONOUS DISTRIBUTED COMPUTING SYSTEM
Michel HURFIN, Noël PLOUZEAU, Michel RAYNAL
Janvier 1992, 14 pages.

PI 631 PROTOCOLES SIMPLES POUR L'IMPLEMENTATION REPARTIE DES SEMAPHORES
Michel RAYNAL
Janvier 1992, 14 pages.

PI 632 L-STABLE PARALLEL ONE-BLOCK METHODS FOR ORDINARY DIFFERENTIAL EQUATIONS
Philippe CHARTIER, Bernard PHILIPPE
Janvier 1992, 28 pages.

PI 633 ON EFFICIENT CHARACTERIZING SOLUTIONS OF LINEAR DIOPHANTINE EQUATIONS AND ITS APPLICATION TO DATA DEPENDENCE ANALYSIS
Christine EISENBEIS, Olivier TEMAM, Harry WIJSHOFF
Janvier 1992, 22 pages.

PI 634 UN NOYAU DE SYSTEME REPARTI POUR LES APPLICATIONS GEREES PAR UN TEMPS VIRTUEL
Philippe INGELS, Carlos MAZIERO, Michel RAYNAL
Janvier 1992, 20 pages.

PI 635 A NOTE ON CHERNIKOVA'S ALGIORITHM
Hervé LE VERGE
Février 1992,28 pages.

PI 636 ENSEIGNER LA TYPOGRAPHIE NUMERIQUE
Jacques ANDRE, Roger D. HERSCH
Février 1992, 26 pages.

PI 637 TRADE-OFFS BETWEEN SHARED VIRTUAL MEMORY AND MESSAGE PASSING ON AN iPSC/2 HYPERCUBE
Thierry PRIOL, Zakaria LAHJOMRI
Février 1992, 26 pages.

PI 638 RUPTURES ET CONTINUITES DANS UN CHANGEMENT DE SYSTEME TECHNIQUE
Alan MARSHALL
Mars 1992, 510 pages.

PI 639 EFFICIENT LINEAR SYSTOLIC ARRAY FOR THE KNAPSACK PROBLEM
Rumen ANDONOV, Patrice QUINTON
Mars 1992, 18 pages.

PI 640 TOWARDS THE RECONSTRUCTION OF POSET
Dieter KRATSCH, Jean-Xavier RAMPON
Mars 1992, 22 pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249-6399